BOOK 20: TYPESCRIPT ADVANCED
Chapter 1: Generics Deep Dive

WHY GENERICS MATTER

Generics create reusable components.

They work with multiple types while keeping type safety.

Write once, use with any type.

GENERIC FUNCTIONS REVIEW

Basic generic function:

```
function identity<T>(value: T): T {
 return value
}

let num = identity<number>(42)
let str = identity<string>("hello")
```

T is a placeholder for any type.

TYPE INFERENCE WITH GENERICS

TypeScript infers type from argument:

```
let num = identity(42)
let str = identity("hello")
```

No need to specify <number> or <string>.

TypeScript figures it out.

MULTIPLE TYPE PARAMETERS

More than one generic type:

```
function pair<T, U>(first: T, second: U): [T, U] {
 return [first, second]
}

let result = pair("Ali", 25)
```

Type is [string, number].

GENERIC CONSTRAINTS

Limit what types can be used:

```
function getLength<T extends { length: number }>(item: T):
number {
 return item.length
}

getLength("hello")
getLength([1, 2, 3])
getLength({ length: 10 })
getLength(42)
```

Error: number has no length.

T must have a length property.

USING KEYOF WITH GENERICS

Access object properties safely:

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
 return obj[key]
}

let user = { name: "Ali", age: 25, email: "ali@example.com" }

let name = getProperty(user, "name")
let age = getProperty(user, "age")
let invalid = getProperty(user, "phone")
```

Error: phone is not a key of user.

GENERIC INTERFACES

Interfaces with type parameters:

```
interface Container<T> {
 value: T;
 getValue(): T;
 setValue(value: T): void;
}

let numberContainer: Container<number> = {
 value: 0,
 getValue() {
 return this.value
 },
 setValue(value) {
 this.value = value
 }
```

```
}

numberContainer.setValue(42)
console.log(numberContainer.getValue())
```

GENERIC CLASSES

Classes with type parameters:

```
class Stack<T> {
 private items: T[] = []

 push(item: T): void {
 this.items.push(item)
 }

 pop(): T | undefined {
 return this.items.pop()
 }

 peek(): T | undefined {
 return this.items[this.items.length - 1]
 }

 isEmpty(): boolean {
 return this.items.length === 0
 }
}

let numberStack = new Stack<number>()
numberStack.push(1)
numberStack.push(2)
numberStack.push(3)

console.log(numberStack.pop())

let stringStack = new Stack<string>()
```

```
stringStack.push("hello")
stringStack.push("world")
```

GENERIC TYPE ALIASES

Type alias with parameters:

```
type Result<T> = {
 success: boolean;
 data: T | null;
 error: string | null;
}

let userResult: Result<{ name: string }> = {
 success: true,
 data: { name: "Ali" },
 error: null
}

let numberResult: Result<number> = {
 success: false,
 data: null,
 error: "Not found"
}
```

DEFAULT TYPE PARAMETERS

Provide default type:

```
interface ApiResponse<T = unknown> {
 data: T;
 status: number;
}
```

```
let response1: ApiResponse<string> = { data: "hello", status: 200
}
let response2: ApiResponse = { data: { any: "thing" }, status: 200 }
```

If no type given, uses unknown.

GENERIC CONSTRAINTS WITH EXTENDS

Complex constraints:

```
interface HasId {
 id: number;
}

interface HasName {
 name: string;
}

function displayEntity<T extends HasId & HasName>(entity: T):
string {
 return entity.id + ": " + entity.name
}

displayEntity({ id: 1, name: "Ali" })
displayEntity({ id: 2, name: "Sara", email: "sara@example.com" })
displayEntity({ name: "Omar" })
```

Error: Missing id.

CONDITIONAL TYPES

Type depends on condition:

```
type IsString<T> = T extends string ? true : false
```

type A = IsString<string>
type B = IsString<number>

A is true.

B is false.

INFER KEYWORD

Extract type from structure:

type ElementType<T> = T extends (infer E)[] ? E : never

type A = ElementType<string[]>
type B = ElementType<number[]>
type C = ElementType<string>

A is string.

B is number.

C is never.

EXTRACTING RETURN TYPE

Get function return type:

type ReturnType<T> = T extends (...args: any[]) => infer R ? R
: never

function greet(): string {
 return "Hello"
}

type GreetReturn = ReturnType<typeof greet>

GreetReturn is string.

EXTRACTING PARAMETER TYPES

Get function parameter types:

type Parameters<T> = T extends (...args: infer P) => any ? P : never

function add(a: number, b: number): number {
 return a + b
}

type AddParams = Parameters<typeof add>

AddParams is [number, number].

DISTRIBUTIVE CONDITIONAL TYPES

Type distributes over union:

type ToArray<T> = T extends any ? T[] : never

type Result = ToArray<string | number>

Result is string[] | number[].

Prevent distribution with brackets:

type ToArrayNonDist<T> = [T] extends [any] ? T[] : never

type Result = ToArrayNonDist<string | number>

Result is (string | number)[].

MAPPED TYPES WITH GENERICS

Transform object types:

```
type Readonly<T> = {
 readonly [K in keyof T]: T[K]
}

type User = {
 name: string;
 age: number;
}

type ReadonlyUser = Readonly<User>
```

All properties become readonly.

Partial makes all optional:

```
type Partial<T> = {
 [K in keyof T]?: T[K]
}

type PartialUser = Partial<User>
```

RECURSIVE GENERICS

Generic that references itself:

```
type DeepReadonly<T> = {
 readonly [K in keyof T]: T[K] extends object ?
DeepReadonly<T[K]> : T[K]
}

type User = {
 name: string;
 address: {
 city: string;
 country: string;
 }
}

type ReadonlyUser = DeepReadonly<User>
```

Nested objects also become readonly.

FACTORY PATTERN WITH GENERICS

Create instances of type:

```
interface Constructor<T> {
 new (...args: any[]): T;
}

function createInstance<T>(ctor: Constructor<T>, ...args: any[]):
T {
 return new ctor(...args)
}

class Person {
 constructor(public name: string) {}
}

let person = createInstance(Person, "Ali")
```

GENERIC REPOSITORY PATTERN

Data access with generics:

```
interface Entity {
 id: number;
}

class Repository<T extends Entity> {
 private items: T[] = []

 add(item: T): void {
 this.items.push(item)
 }

 findById(id: number): T | undefined {
 return this.items.find(item => item.id === id)
 }

 findAll(): T[] {
 return [...this.items]
 }

 update(id: number, updates: Partial<T>): T | undefined {
 let item = this.findById(id)
 if (item) {
 Object.assign(item, updates)
 }
 return item
 }

 delete(id: number): boolean {
 let index = this.items.findIndex(item => item.id === id)
 if (index !== -1) {
 this.items.splice(index, 1)
 return true
 }
```

```
 return false
 }
}

interface User extends Entity {
 name: string;
 email: string;
}

let userRepo = new Repository<User>()
userRepo.add({ id: 1, name: "Ali", email: "ali@example.com" })
userRepo.add({ id: 2, name: "Sara", email: "sara@example.com" })

let user = userRepo.findById(1)
userRepo.update(1, { name: "Ali Updated" })
```

PRACTICE

Create generic functions with constraints.

Build a generic class.

Use conditional types.

Practice with infer keyword.

Create a generic data structure.

Implement a factory pattern.

WHAT YOU LEARNED

You learned advanced generic patterns.

You learned generic constraints.

You learned conditional types and infer.

You learned distributive types.

You learned practical generic applications.

The next chapter teaches classes with types.

BOOK 20: TYPESCRIPT ADVANCED

Chapter 2: Classes with Types

CLASS BASICS

TypeScript classes with typed members:

```
class Person {
 name: string
 age: number

 constructor(name: string, age: number) {
 this.name = name
 this.age = age
 }

 greet(): string {
 return "Hello, I am " + this.name
 }
}

let person = new Person("Ali", 25)
console.log(person.greet())
```

PARAMETER PROPERTIES

Shorter syntax for common pattern:

```
class Person {
 constructor(public name: string, public age: number) {}
}

let person = new Person("Ali", 25)
console.log(person.name)
```

public in constructor creates and assigns property.

ACCESS MODIFIERS

public (default):

Accessible everywhere.

```
class Person {
 public name: string

 constructor(name: string) {
 this.name = name
 }
}

let person = new Person("Ali")
console.log(person.name)
```

private:

Only accessible inside class.

```
class Person {
```

```
 private id: number

 constructor(id: number) {
 this.id = id
 }

 getId(): number {
 return this.id
 }
}

let person = new Person(123)
console.log(person.id)
```

Error: id is private.

protected:

Accessible in class and subclasses.

```
class Person {
 protected name: string

 constructor(name: string) {
 this.name = name
 }
}

class Employee extends Person {
 displayName(): string {
 return this.name
 }
}
```

READONLY PROPERTIES

Cannot be changed after initialization:

```
class Person {
 readonly id: number

 constructor(id: number) {
 this.id = id
 }
}

let person = new Person(123)
person.id = 456
```

Error: Cannot assign to readonly.

GETTERS AND SETTERS

Controlled property access:

```
class Person {
 private _age: number = 0

 get age(): number {
 return this._age
 }

 set age(value: number) {
 if (value < 0) {
 throw new Error("Age cannot be negative")
 }
 this._age = value
 }
}

let person = new Person()
person.age = 25
console.log(person.age)
```

person.age = -5

Error thrown.

STATIC MEMBERS

Belong to class not instances:

```
class Counter {
 static count: number = 0

 constructor() {
 Counter.count++
 }

 static getCount(): number {
 return Counter.count
 }
}

new Counter()
new Counter()
new Counter()

console.log(Counter.getCount())
```

ABSTRACT CLASSES

Cannot be instantiated directly:

```
abstract class Shape {
 abstract area(): number
 abstract perimeter(): number
```

```
 describe(): string {
 return "A shape with area " + this.area()
 }
}

class Rectangle extends Shape {
 constructor(private width: number, private height: number) {
 super()
 }

 area(): number {
 return this.width * this.height
 }

 perimeter(): number {
 return 2 * (this.width + this.height)
 }
}

class Circle extends Shape {
 constructor(private radius: number) {
 super()
 }

 area(): number {
 return Math.PI * this.radius * this.radius
 }

 perimeter(): number {
 return 2 * Math.PI * this.radius
 }
}

let rect = new Rectangle(10, 5)
console.log(rect.describe())
```

IMPLEMENTING INTERFACES

Class follows interface contract:

```
interface Printable {
 print(): void
}

interface Serializable {
 serialize(): string
}

class Document implements Printable, Serializable {
 constructor(private content: string) {}

 print(): void {
 console.log(this.content)
 }

 serialize(): string {
 return JSON.stringify({ content: this.content })
 }
}

let doc = new Document("Hello World")
doc.print()
console.log(doc.serialize())
```

CLASS AS INTERFACE

Class can be used as type:

```
class Point {
 x: number = 0
 y: number = 0
}

let point: Point = { x: 10, y: 20 }
```

Object literal matches class shape.

INHERITANCE

Extend existing class:

```
class Animal {
 constructor(public name: string) {}

 move(): void {
 console.log(this.name + " is moving")
 }
}

class Dog extends Animal {
 bark(): void {
 console.log(this.name + " barks")
 }

 move(): void {
 console.log(this.name + " runs")
 }
}

let dog = new Dog("Rex")
dog.bark()
dog.move()
```

SUPER KEYWORD

Call parent class:

```
class Animal {
```

```
 constructor(public name: string) {}

 speak(): string {
 return this.name + " makes a sound"
 }
}

class Dog extends Animal {
 constructor(name: string, public breed: string) {
 super(name)
 }

 speak(): string {
 return super.speak() + " - woof!"
 }
}

let dog = new Dog("Rex", "German Shepherd")
console.log(dog.speak())
```

PRIVATE FIELDS (ECMAScript)

True private with hash:

```
class Person {
 #ssn: string

 constructor(ssn: string) {
 this.#ssn = ssn
 }

 getSSN(): string {
 return this.#ssn
 }
}

let person = new Person("123-45-6789")
```

```
console.log(person.#ssn)
```

Error: Private field.

Cannot access from outside at all.

THIS TYPE

Return current class type:

```
class Builder {
 private value: number = 0

 setValue(value: number): this {
 this.value = value
 return this
 }

 getValue(): number {
 return this.value
 }
}

class AdvancedBuilder extends Builder {
 double(): this {
 return this.setValue(this.getValue() * 2)
 }
}

let result = new AdvancedBuilder()
 .setValue(10)
 .double()
 .getValue()

console.log(result)
```

GENERIC CLASSES

Class with type parameter:

```
class Box<T> {
 private content: T | undefined

 set(value: T): void {
 this.content = value
 }

 get(): T | undefined {
 return this.content
 }

 hasContent(): boolean {
 return this.content !== undefined
 }
}

let numberBox = new Box<number>()
numberBox.set(42)
console.log(numberBox.get())

let stringBox = new Box<string>()
stringBox.set("hello")
console.log(stringBox.get())
```

CLASS DECORATORS

Modify classes (experimental):

```
function sealed(constructor: Function) {
 Object.seal(constructor)
 Object.seal(constructor.prototype)
```

```
}

@sealed
class Greeter {
 greeting: string

 constructor(message: string) {
 this.greeting = message
 }
}
```

Requires experimentalDecorators in tsconfig.

METHOD DECORATORS

Modify methods:

```
function log(
 target: any,
 propertyKey: string,
 descriptor: PropertyDescriptor
) {
 let original = descriptor.value

 descriptor.value = function(...args: any[]) {
 console.log("Calling " + propertyKey + " with", args)
 return original.apply(this, args)
 }
}

class Calculator {
 @log
 add(a: number, b: number): number {
 return a + b
 }
}
```

```
let calc = new Calculator()
calc.add(2, 3)
```

SINGLETON PATTERN

Only one instance:

```
class Database {
 private static instance: Database

 private constructor() {}

 static getInstance(): Database {
 if (!Database.instance) {
 Database.instance = new Database()
 }
 return Database.instance
 }

 query(sql: string): void {
 console.log("Executing: " + sql)
 }
}

let db1 = Database.getInstance()
let db2 = Database.getInstance()

console.log(db1 === db2)
```

FACTORY PATTERN

Create instances through method:

```
abstract class Vehicle {
```

```
 abstract drive(): void
}

class Car extends Vehicle {
 drive(): void {
 console.log("Driving a car")
 }
}

class Motorcycle extends Vehicle {
 drive(): void {
 console.log("Riding a motorcycle")
 }
}

class VehicleFactory {
 static create(type: "car" | "motorcycle"): Vehicle {
 switch (type) {
 case "car":
 return new Car()
 case "motorcycle":
 return new Motorcycle()
 }
 }
}

let vehicle = VehicleFactory.create("car")
vehicle.drive()
```

MIXINS

Combine behaviors:

```
type Constructor<T = {}> = new (...args: any[]) => T

function Timestamped<TBase extends Constructor>(Base: TBase) {
 return class extends Base {
```

```
 timestamp = new Date()
 }
}

function Activatable<TBase extends Constructor>(Base: TBase) {
 return class extends Base {
 isActive = false

 activate(): void {
 this.isActive = true
 }

 deactivate(): void {
 this.isActive = false
 }
 }
}

class User {
 constructor(public name: string) {}
}

let TimestampedActivatableUser =
Timestamped(Activatable(User))

let user = new TimestampedActivatableUser("Ali")
console.log(user.timestamp)
user.activate()
console.log(user.isActive)
```

PRACTICE

Create classes with access modifiers.

Implement interfaces.

Use abstract classes.

Build generic classes.

Practice inheritance and super.

Implement design patterns.

WHAT YOU LEARNED

You learned typed class properties.

You learned access modifiers.

You learned abstract classes and interfaces.

You learned generic classes.

You learned design patterns with classes.

The next chapter teaches type narrowing.

BOOK 20: TYPESCRIPT ADVANCED

Chapter 3: Type Narrowing

WHAT IS TYPE NARROWING

Type narrowing refines a broad type to a specific one.

Start with union type.

Use checks to narrow to single type.

TypeScript tracks the narrowed type.

TYPEOF GUARDS

Check primitive types:

```
function printValue(value: string | number) {
 if (typeof value === "string") {
 console.log(value.toUpperCase())
 } else {
 console.log(value.toFixed(2))
 }
}

printValue("hello")
printValue(42.567)
```

Inside if block, value is string.

In else block, value is number.

TYPEOF TYPE LIST

typeof works with:

string
number
boolean
symbol
undefined
object
function
bigint

Example:

```
function process(value: string | number | boolean | undefined) {
 if (typeof value === "string") {
 return value.length
 }
 if (typeof value === "number") {
 return value * 2
 }
 if (typeof value === "boolean") {
 return value ? "yes" : "no"
 }
 return "undefined"
}
```

TRUTHINESS NARROWING

Check if value exists:

```
function printName(name: string | null | undefined) {
 if (name) {
 console.log(name.toUpperCase())
 } else {
 console.log("No name provided")
 }
}
```

Truthy check narrows out null and undefined.

Watch for falsy values:

```
function printLength(value: string | null) {
 if (value) {
 console.log(value.length)
 }
}
```

printLength("")

Empty string is falsy.

May not be what you want.

Better:

```
function printLength(value: string | null) {
 if (value !== null) {
 console.log(value.length)
 }
}
```

EQUALITY NARROWING

Check for specific values:

```
function compare(a: string | number, b: string | boolean) {
 if (a === b) {
 console.log(a.toUpperCase())
 console.log(b.toUpperCase())
 }
}
```

If a equals b, both must be string.

Equality with null:

```
function printValue(value: string | null | undefined) {
 if (value != null) {
 console.log(value)
 }
}
```

!= null also excludes undefined.

IN OPERATOR NARROWING

Check if property exists:

```
type Fish = { swim: () => void }
type Bird = { fly: () => void }

function move(animal: Fish | Bird) {
 if ("swim" in animal) {
 animal.swim()
 } else {
 animal.fly()
 }
}
```

If swim property exists, it is Fish.

INSTANCEOF NARROWING

Check constructor:

```
function logDate(value: Date | string) {
 if (value instanceof Date) {
 console.log(value.toISOString())
 } else {
 console.log(value)
 }
}

logDate(new Date())
logDate("2024-01-01")
```

TYPE PREDICATES

Custom type guard function:

```
function isFish(animal: Fish | Bird): animal is Fish {
 return (animal as Fish).swim !== undefined
}

function move(animal: Fish | Bird) {
 if (isFish(animal)) {
 animal.swim()
 } else {
 animal.fly()
 }
}
```

animal is Fish is type predicate.

Returns boolean.

Tells TypeScript what type if true.

ASSERTION FUNCTIONS

Assert type or throw:

```
function assertIsString(value: unknown): asserts value is string {
 if (typeof value !== "string") {
 throw new Error("Expected string")
 }
}

function processValue(value: unknown) {
 assertIsString(value)
```

```
 console.log(value.toUpperCase())
}
```

After assertIsString, value is string.

DISCRIMINATED UNIONS

Common property distinguishes types:

```
type Circle = {
 kind: "circle";
 radius: number;
}

type Square = {
 kind: "square";
 size: number;
}

type Shape = Circle | Square

function getArea(shape: Shape): number {
 switch (shape.kind) {
 case "circle":
 return Math.PI * shape.radius * shape.radius
 case "square":
 return shape.size * shape.size
 }
}

let circle: Circle = { kind: "circle", radius: 5 }
let square: Square = { kind: "square", size: 4 }

console.log(getArea(circle))
console.log(getArea(square))
```

EXHAUSTIVE CHECKS

Ensure all cases handled:

```
type Shape = Circle | Square | Triangle

type Triangle = {
 kind: "triangle";
 base: number;
 height: number;
}

function getArea(shape: Shape): number {
 switch (shape.kind) {
 case "circle":
 return Math.PI * shape.radius * shape.radius
 case "square":
 return shape.size * shape.size
 case "triangle":
 return 0.5 * shape.base * shape.height
 default:
 let exhaustiveCheck: never = shape
 return exhaustiveCheck
 }
}
```

If new type added but not handled, error on never.

CONTROL FLOW ANALYSIS

TypeScript tracks types through code:

```
function example(value: string | number | null) {
 if (value === null) {
 console.log("Is null")
```

```
 return
 }

 if (typeof value === "string") {
 console.log(value.toUpperCase())
 return
 }

 console.log(value.toFixed(2))
}
```

At each point, TypeScript knows remaining types.

NARROWING WITH NEVER

Impossible cases become never:

```
type A = string & number
```

A is never.

No value can be both string and number.

OPTIONAL CHAINING

Safe property access:

```
type User = {
 name: string;
 address?: {
 city: string;
 zip?: string;
 }
}
```

```
function getCity(user: User): string | undefined {
 return user.address?.city
}

function getZip(user: User): string | undefined {
 return user.address?.zip
}
```

Returns undefined if path does not exist.

NULLISH COALESCING

Default for null or undefined:

```
function getDisplayName(name: string | null | undefined): string {
 return name ?? "Anonymous"
}

getDisplayName("Ali")
getDisplayName(null)
getDisplayName(undefined)
```

Only null and undefined trigger default.

Empty string does not.

NON-NULL ASSERTION

Tell TypeScript value exists:

```
function process(value: string | null) {
 let length = value!.length
}
```

Exclamation mark asserts not null.

Use carefully.

Can cause runtime errors.

ARRAY NARROWING

Check array contents:

```
function processArray(arr: (string | number)[]) {
 let strings = arr.filter((item): item is string => {
 return typeof item === "string"
 })

 strings.forEach(s => console.log(s.toUpperCase()))
}
```

Type predicate in filter.

OBJECT NARROWING

Check object shape:

```
type Dog = { bark: () => void }
type Cat = { meow: () => void }
type Animal = Dog | Cat

function isDog(animal: Animal): animal is Dog {
 return "bark" in animal
}

function makeSound(animal: Animal) {
```

```
 if (isDog(animal)) {
 animal.bark()
 } else {
 animal.meow()
 }
}
```

BRANDED TYPES

Distinguish same primitive types:

```
type UserId = string & { _brand: "UserId" }
type ProductId = string & { _brand: "ProductId" }

function createUserId(id: string): UserId {
 return id as UserId
}

function createProductId(id: string): ProductId {
 return id as ProductId
}

function getUser(id: UserId) {
 console.log("Getting user: " + id)
}

let userId = createUserId("user123")
let productId = createProductId("product456")

getUser(userId)
getUser(productId)
```

Error: ProductId not assignable to UserId.

PRACTICE

Use typeof for primitive narrowing.

Create type predicate functions.

Use discriminated unions.

Implement exhaustive checks.

Practice with optional chaining.

Create branded types.

WHAT YOU LEARNED

You learned typeof and instanceof guards.

You learned truthiness and equality narrowing.

You learned type predicates.

You learned discriminated unions.

You learned exhaustive checking.

The next chapter teaches utility types.
BOOK 20: TYPESCRIPT ADVANCED
Chapter 4: Utility Types

WHAT ARE UTILITY TYPES

Built-in types that transform other types.

Create new types from existing ones.

Save time with common transformations.

PARTIAL

Make all properties optional:

```
interface User {
 name: string;
 email: string;
 age: number;
}

type PartialUser = Partial<User>
```

PartialUser is:

```
{
 name?: string;
 email?: string;
 age?: number;
}
```

Use case - update functions:

```
function updateUser(user: User, updates: Partial<User>): User {
 return { ...user, ...updates }
}

let user: User = { name: "Ali", email: "ali@example.com", age: 25 }
let updated = updateUser(user, { age: 26 })
```

REQUIRED

Make all properties required:

```
interface Config {
 host?: string;
 port?: number;
 secure?: boolean;
}

type RequiredConfig = Required<Config>
```

RequiredConfig is:

```
{
 host: string;
 port: number;
 secure: boolean;
}
```

Use case - validate config:

```
function validateConfig(config: Config): RequiredConfig {
 return {
 host: config.host ?? "localhost",
 port: config.port ?? 3000,
 secure: config.secure ?? false
 }
}
```

READONLY

Make all properties readonly:

```
interface User {
```

```
 name: string;
 email: string;
}

type ReadonlyUser = Readonly<User>

ReadonlyUser is:

{
 readonly name: string;
 readonly email: string;
}


Use case - immutable data:

function freezeUser(user: User): ReadonlyUser {
 return Object.freeze(user)
}

let user = freezeUser({ name: "Ali", email: "ali@example.com" })
user.name = "Sara"

Error: Cannot assign to readonly.




RECORD

Create object type with specific keys:

type UserRoles = "admin" | "editor" | "viewer"

type RolePermissions = Record<UserRoles, string[]>

let permissions: RolePermissions = {
 admin: ["read", "write", "delete"],
 editor: ["read", "write"],
 viewer: ["read"]
```

}

Dynamic keys:

type StringMap = Record<string, string>

let translations: StringMap = {
 hello: "Hola",
 goodbye: "Adios"
}

PICK

Select specific properties:

interface User {
 id: number;
 name: string;
 email: string;
 password: string;
 createdAt: Date;
}

type PublicUser = Pick<User, "id" | "name" | "email">

PublicUser is:

{
 id: number;
 name: string;
 email: string;
}

Use case - public API:

```
function getPublicUser(user: User): PublicUser {
 return {
 id: user.id,
 name: user.name,
 email: user.email
 }
}
```

OMIT

Remove specific properties:

```
interface User {
 id: number;
 name: string;
 email: string;
 password: string;
}

type SafeUser = Omit<User, "password">
```

SafeUser is:

```
{
 id: number;
 name: string;
 email: string;
}
```

Omit multiple:

```
type PublicUser = Omit<User, "password" | "email">
```

EXCLUDE

Remove types from union:

type AllColors = "red" | "green" | "blue" | "yellow"

type PrimaryColors = Exclude<AllColors, "yellow">

PrimaryColors is "red" | "green" | "blue".

Multiple exclusions:

type JustRed = Exclude<AllColors, "green" | "blue" | "yellow">

EXTRACT

Keep only matching types:

type AllTypes = string | number | boolean | null

type Primitives = Extract<AllTypes, string | number>

Primitives is string | number.

NONNULLABLE

Remove null and undefined:

type MaybeString = string | null | undefined

type DefiniteString = NonNullable<MaybeString>

DefiniteString is string.

RETURNTYPE

Get function return type:

```
function getUser() {
 return { name: "Ali", age: 25 }
}

type User = ReturnType<typeof getUser>
```

User is { name: string; age: number }.

With generic functions:

```
function identity<T>(value: T): T {
 return value
}

type Result = ReturnType<typeof identity<string>>
```

PARAMETERS

Get function parameter types:

```
function greet(name: string, age: number): string {
 return name + " is " + age
}

type GreetParams = Parameters<typeof greet>
```

GreetParams is [string, number].

Access individual parameter:

type FirstParam = Parameters<typeof greet>[0]

CONSTRUCTORPARAMETERS

Get constructor parameter types:

class Person {
 constructor(public name: string, public age: number) {}
}

type PersonParams = ConstructorParameters<typeof Person>

PersonParams is [string, number].

INSTANCETYPE

Get instance type from constructor:

class Person {
 constructor(public name: string) {}
}

type PersonInstance = InstanceType<typeof Person>

PersonInstance is Person.

THISPARAMETERTYPE

Get type of this:

```
function greet(this: { name: string }) {
 return "Hello " + this.name
}

type GreetThis = ThisParameterType<typeof greet>
```

OMITTHISPARAMETER

Remove this from function:

```
function greet(this: { name: string }): string {
 return "Hello " + this.name
}

type GreetFn = OmitThisParameter<typeof greet>
```

GreetFn is () => string.

AWAITED

Unwrap Promise type:

```
type PromiseString = Promise<string>

type Unwrapped = Awaited<PromiseString>
```

Unwrapped is string.

Nested promises:

```
type Nested = Promise<Promise<Promise<number>>>
```

```
type Result = Awaited<Nested>

Result is number.
```

UPPERCASE AND LOWERCASE

Transform string literal types:

```
type Greeting = "hello"

type Upper = Uppercase<Greeting>
type Lower = Lowercase<Greeting>

Upper is "HELLO".

Lower is "hello".
```

CAPITALIZE AND UNCAPITALIZE

Change first letter:

```
type Word = "hello"

type Capitalized = Capitalize<Word>
type Uncapitalized = Uncapitalize<"Hello">

Capitalized is "Hello".

Uncapitalized is "hello".
```

COMBINING UTILITY TYPES

Chain utilities:

```
interface User {
 id: number;
 name: string;
 email: string;
 password: string;
}

type CreateUserInput = Omit<Partial<User>, "id">
```

CreateUserInput is:

```
{
 name?: string;
 email?: string;
 password?: string;
}
```

More complex:

```
type ReadonlyPick<T, K extends keyof T> = Readonly<Pick<T, K>>

type PublicReadonlyUser = ReadonlyPick<User, "id" | "name">
```

CUSTOM UTILITY TYPES

Create your own:

```
type Nullable<T> = T | null

type Optional<T> = T | undefined
```

```
type Maybe<T> = T | null | undefined
```

Deep readonly:

```
type DeepReadonly<T> = {
 readonly [K in keyof T]: T[K] extends object
 ? DeepReadonly<T[K]>
 : T[K]
}
```

Make specific properties optional:

```
type PartialBy<T, K extends keyof T> = Omit<T, K> &
Partial<Pick<T, K>>

type User = {
 id: number;
 name: string;
 email: string;
}

type UserWithOptionalEmail = PartialBy<User, "email">
```

PRACTICE

Use Partial for update functions.

Use Pick and Omit to shape types.

Use Record for dictionaries.

Use ReturnType with functions.

Combine multiple utility types.

Create custom utility types.

WHAT YOU LEARNED

You learned Partial, Required, Readonly.

You learned Record, Pick, Omit.

You learned Exclude, Extract, NonNullable.

You learned ReturnType and Parameters.

You learned to combine and create utilities.

The next chapter teaches advanced types.

BOOK 20: TYPESCRIPT ADVANCED

Chapter 5: Advanced Types

MAPPED TYPES

Transform every property in a type:

```
type Readonly<T> = {
 readonly [K in keyof T]: T[K]
}
```

K in keyof T iterates over all keys.

T[K] gets the type of each property.

Custom mapped type:

```
type Optional<T> = {
 [K in keyof T]?: T[K]
}

type Nullable<T> = {
 [K in keyof T]: T[K] | null
}
```

KEY REMAPPING

Change key names:

```
type Getters<T> = {
 [K in keyof T as `get${Capitalize<string & K>}`]: () => T[K]
}

interface Person {
 name: string;
 age: number;
}

type PersonGetters = Getters<Person>
```

PersonGetters is:

```
{
 getName: () => string;
 getAge: () => number;
}
```

FILTERING KEYS

Keep only some properties:

```
type OnlyStrings<T> = {
 [K in keyof T as T[K] extends string ? K : never]: T[K]
}

interface Mixed {
 name: string;
 age: number;
 email: string;
}

type StringProps = OnlyStrings<Mixed>
```

StringProps is:

```
{
 name: string;
 email: string;
}
```

CONDITIONAL TYPES

Type depends on condition:

```
type IsString<T> = T extends string ? true : false

type A = IsString<string>
type B = IsString<number>
```

A is true.

B is false.

CONDITIONAL TYPE PATTERNS

Check for specific type:

type IsArray<T> = T extends any[] ? true : false

Extract array element:

type ArrayElement<T> = T extends (infer E)[] ? E : never

Function return type:

type ReturnOf<T> = T extends (...args: any[]) => infer R ? R : never

Promise unwrap:

type Unwrap<T> = T extends Promise<infer U> ? U : T

DISTRIBUTIVE CONDITIONAL TYPES

Condition distributes over union:

type ToArray<T> = T extends any ? T[] : never

type Result = ToArray<string | number>

Result is string[] | number[].

Each union member processed separately.

Prevent distribution:

type ToArrayNonDist<T> = [T] extends [any] ? T[] : never

type Result = ToArrayNonDist<string | number>

Result is (string | number)[].

INFER KEYWORD

Extract type from pattern:

type GetReturnType<T> = T extends (...args: any[]) => infer R
? R : never

type R1 = GetReturnType<() => string>
type R2 = GetReturnType<(x: number) => boolean>

R1 is string.

R2 is boolean.

Multiple infer:

type FirstArg<T> = T extends (first: infer F, ...args: any[]) =>
any
 ? F
 : never

type F1 = FirstArg<(a: string, b: number) => void>

F1 is string.

TEMPLATE LITERAL TYPES

String type patterns:

```
type EventName = `on${string}`

let valid: EventName = "onClick"
let invalid: EventName = "click"
```

Error: click does not match pattern.

With unions:

```
type Verb = "get" | "set"
type Property = "Name" | "Age"

type Method = `${Verb}${Property}`
```

Method is "getName" | "getAge" | "setName" | "setAge".

TEMPLATE LITERAL UTILITIES

Built-in string manipulations:

```
type Upper = Uppercase<"hello">
type Lower = Lowercase<"HELLO">
type Cap = Capitalize<"hello">
type Uncap = Uncapitalize<"Hello">
```

Upper is "HELLO".

Lower is "hello".

Cap is "Hello".

Uncap is "hello".

RECURSIVE TYPES

Type references itself:

```
type JSONValue =
 | string
 | number
 | boolean
 | null
 | JSONValue[]
 | { [key: string]: JSONValue }
```

Deep partial:

```
type DeepPartial<T> = {
 [K in keyof T]?: T[K] extends object ? DeepPartial<T[K]> : T[K]
}
```

Nested path:

```
type Paths<T, K extends keyof T = keyof T> =
 K extends string
 ? T[K] extends object
 ? K | `${K}.${Paths<T[K]>}`
 : K
 : never

interface User {
 name: string;
 address: {
 city: string;
 zip: string;
 }
}

type UserPaths = Paths<User>
```

UserPaths is "name" | "address" | "address.city" | "address.zip".

TYPE PREDICATES

Narrow types with functions:

```
function isString(value: unknown): value is string {
 return typeof value === "string"
}

function process(value: unknown) {
 if (isString(value)) {
 console.log(value.toUpperCase())
 }
}
```

ASSERTION SIGNATURES

Assert type or throw:

```
function assertIsNumber(value: unknown): asserts value is number {
 if (typeof value !== "number") {
 throw new Error("Not a number")
 }
}

function double(value: unknown): number {
 assertIsNumber(value)
 return value * 2
}
```

VARIANCE ANNOTATIONS

Control type compatibility:

```
type Getter<out T> = () => T
type Setter<in T> = (value: T) => void
type GetSet<in out T> = {
 get: () => T;
 set: (value: T) => void;
}
```

out = covariant (output position).

in = contravariant (input position).

in out = invariant.

INDEX ACCESS TYPES

Access type by key:

```
interface User {
 name: string;
 age: number;
 address: {
 city: string;
 }
}

type Name = User["name"]
type Age = User["age"]
type City = User["address"]["city"]
```

With union:

type NameOrAge = User["name" | "age"]

NameOrAge is string | number.

All properties:

type AllTypes = User[keyof User]

KEYOF TYPE OPERATOR

Get union of keys:

```
interface User {
 name: string;
 age: number;
}
```

type UserKeys = keyof User

UserKeys is "name" | "age".

TYPEOF TYPE OPERATOR

Get type from value:

let user = { name: "Ali", age: 25 }

type User = typeof user

User is { name: string; age: number }.

With const assertion:

```
let colors = ["red", "green", "blue"] as const

type Colors = typeof colors
```

Colors is readonly ["red", "green", "blue"].

SATISFIES OPERATOR

Check type while preserving inference:

```
type Colors = Record<string, string | number[]>

let colors = {
 red: "#ff0000",
 green: "#00ff00",
 blue: [0, 0, 255]
} satisfies Colors

colors.red.toUpperCase()
```

Type is inferred precisely but validated.

CONST TYPE PARAMETERS

Keep literal types:

```
function identity<const T>(value: T): T {
 return value
}

let result = identity({ x: 1, y: 2 })
```

result is { readonly x: 1; readonly y: 2 }.

PRACTICE

Create mapped types.

Use conditional types with infer.

Build template literal types.

Create recursive types.

Use type predicates.

Practice index access types.

WHAT YOU LEARNED

You learned mapped types.

You learned conditional types.

You learned template literal types.

You learned recursive types.

You learned advanced type operators.

The next chapter teaches modules and declaration files.

TYPESCRIPT MODULES

TypeScript uses ES modules:

```
export function greet(name: string): string {
 return "Hello " + name
}

export interface User {
 name: string;
 age: number;
}
```

Import:

```
import { greet, User } from "./utils"

let user: User = { name: "Ali", age: 25 }
console.log(greet(user.name))
```

EXPORT TYPES

Default export:

```
export default class User {
 constructor(public name: string) {}
}
```

Named exports:

```
export const VERSION = "1.0.0"
export type ID = string | number
export interface Config {
 debug: boolean;
```

}

Re-export:

export { greet } from "./greet"
export { User } from "./user"

Export all:

export * from "./utils"

IMPORT TYPES

Import type only:

import type { User } from "./types"

Only for type information.

Removed during compilation.

Mixed import:

import { greet, type User } from "./utils"

greet is value, User is type only.

MODULE RESOLUTION

TypeScript finds modules:

node - Like Node.js resolution.

bundler - For modern bundlers.

Configure in tsconfig:

```
{
 "compilerOptions": {
 "moduleResolution": "bundler"
 }
}
```

PATH MAPPING

Aliases for imports:

```
{
 "compilerOptions": {
 "baseUrl": ".",
 "paths": {
 "@utils/*": ["src/utils/*"],
 "@types/*": ["src/types/*"]
 }
 }
}
```

Usage:

```
import { greet } from "@utils/greet"
import type { User } from "@types/user"
```

DECLARATION FILES

.d.ts files contain only types.

No implementation.

user.d.ts:

```
export interface User {
 id: number;
 name: string;
 email: string;
}

export function createUser(name: string): User;

export const VERSION: string;
```

WRITING DECLARATION FILES

For existing JavaScript:

math.js:

```
function add(a, b) {
 return a + b;
}

function subtract(a, b) {
 return a - b;
}

module.exports = { add, subtract };
```

math.d.ts:

```
export function add(a: number, b: number): number;
export function subtract(a: number, b: number): number;
```

GLOBAL DECLARATIONS

Types available everywhere:

global.d.ts:

```
declare global {
 interface Window {
 myApp: {
 version: string;
 init(): void;
 }
 }

 function myGlobalFunction(): void;
}

export {}
```

Usage:

```
window.myApp.init()
myGlobalFunction()
```

AMBIENT DECLARATIONS

Declare existing variables:

```
declare const API_URL: string;
```

declare const DEBUG: boolean;

Tell TypeScript these exist at runtime.

MODULE AUGMENTATION

Add to existing module:

```
import "express"

declare module "express" {
 interface Request {
 user?: {
 id: string;
 name: string;
 }
 }
}
```

Usage:

```
app.get("/", (req, res) = > {
 console.log(req.user?.name)
})
```

NAMESPACE MERGING

Add to namespace:

```
declare namespace NodeJS {
 interface ProcessEnv {
 NODE_ENV: "development" | "production";
 PORT: string;
```

```
 DATABASE_URL: string;
 }
}
```

Usage:

```
let env = process.env.NODE_ENV
let port = process.env.PORT
```

DEFINITELYTYPED

Community type definitions.

Install with @types prefix:

```
npm install @types/lodash
npm install @types/express
npm install @types/node
```

Automatic usage:

```
import _ from "lodash"

_.map([1, 2, 3], n => n * 2)
```

Types work automatically.

CREATING TYPES FOR LIBRARIES

Package with no types:

mylib.d.ts:

```
declare module "mylib" {
 export function doSomething(input: string): string;
 export interface Options {
 timeout: number;
 retry: boolean;
 }
 export default function init(options: Options): void;
}
```

TYPE DECLARATION OPTIONS

In tsconfig:

```
{
 "compilerOptions": {
 "declaration": true,
 "declarationDir": "./types",
 "declarationMap": true
 }
}
```

declaration - Generate .d.ts files.

declarationDir - Output directory.

declarationMap - Generate source maps.

TRIPLE SLASH DIRECTIVES

Reference other declarations:

mylib.d.ts:

```
/// <reference types="node" />
/// <reference path="./other.d.ts" />

export function readFile(path: string): Buffer;
```

IMPORT ASSERTIONS

Import JSON and other:

```
import config from "./config.json" assert { type: "json" };

let port = config.port;
```

Configure:

```
{
 "compilerOptions": {
 "resolveJsonModule": true
 }
}
```

EXPORT PATTERNS

Barrel exports:

index.ts:

```
export * from "./user"
export * from "./product"
export * from "./order"
```

Namespace export:

export * as utils from "./utils"

Import as namespace:

import * as validators from "./validators"

validators.isEmail("test@example.com")

ISOLATEDMODULES

Each file is separate module:

```
{
 "compilerOptions": {
 "isolatedModules": true
 }
}
```

Required for some bundlers.

Restrictions:

Cannot use const enum.

Cannot re-export types.

Must export something.

VERBATIM MODULE SYNTAX

Keep import/export as written:

```
{
 "compilerOptions": {
 "verbatimModuleSyntax": true
 }
}
```

Type imports must use import type.

PRACTICE

Create modules with exports.

Use path mapping.

Write declaration files.

Augment existing modules.

Install and use @types packages.

Configure module resolution.

WHAT YOU LEARNED

You learned TypeScript module syntax.

You learned declaration files.

You learned module augmentation.

You learned DefinitelyTyped.

You learned configuration options.

The next chapter teaches TypeScript with React.

BOOK 20: TYPESCRIPT ADVANCED

Chapter 7: TypeScript with React

SETTING UP

Create React with TypeScript:

```
npx create-react-app my-app --template typescript
```

Or with Vite:

```
npm create vite@latest my-app -- --template react-ts
```

Files use .tsx extension.

TYPING FUNCTION COMPONENTS

Basic component:

```
function Greeting({ name }: { name: string }) {
 return <h1>Hello, {name}</h1>
}
```

With interface:

```
interface GreetingProps {
```

```
 name: string;
 age?: number;
}

function Greeting({ name, age }: GreetingProps) {
 return (
 <div>
 <h1>Hello, {name}</h1>
 {age && <p>Age: {age}</p>}
 </div>
 )
}
```

REACT.FC TYPE

Alternate component typing:

```
import { FC } from "react"

interface Props {
 name: string;
}

const Greeting: FC<Props> = ({ name }) => {
 return <h1>Hello, {name}</h1>
}
```

FC automatically types children.

Many prefer explicit props typing.

CHILDREN PROP

Type children explicitly:

```
interface Props {
 children: React.ReactNode;
}

function Container({ children }: Props) {
 return <div className="container">{children}</div>
}
```

Or specific children:

```
interface Props {
 children: React.ReactElement;
}

interface Props {
 children: string;
}
```

TYPING EVENTS

Event handlers:

```
function Form() {
 function handleSubmit(event:
React.FormEvent<HTMLFormElement>) {
 event.preventDefault()
 console.log("Submitted")
 }

 function handleChange(event:
React.ChangeEvent<HTMLInputElement>) {
 console.log(event.target.value)
 }

 return (
```

```
 <form onSubmit={handleSubmit}>
 <input onChange={handleChange} />
 <button type="submit">Submit</button>
 </form>
 )
}
```

COMMON EVENT TYPES

React.MouseEvent<HTMLButtonElement>

React.KeyboardEvent<HTMLInputElement>

React.ChangeEvent<HTMLInputElement>

React.FormEvent<HTMLFormElement>

React.FocusEvent<HTMLInputElement>

React.DragEvent<HTMLDivElement>

TYPING USESTATE

Type inference works:

const [count, setCount] = useState(0)

TypeScript knows count is number.

Explicit type:

const [user, setUser] = useState<User | null>(null)

Complex types:

```
interface State {
 items: string[];
 loading: boolean;
}

const [state, setState] = useState<State>({
 items: [],
 loading: false
})
```

TYPING USEREDUCER

Define action types:

```
type Action =
 | { type: "increment" }
 | { type: "decrement" }
 | { type: "reset"; payload: number }

interface State {
 count: number;
}

function reducer(state: State, action: Action): State {
 switch (action.type) {
 case "increment":
 return { count: state.count + 1 }
 case "decrement":
 return { count: state.count - 1 }
 case "reset":
 return { count: action.payload }
 }
}
```

```
function Counter() {
 const [state, dispatch] = useReducer(reducer, { count: 0 })

 return (
 <div>
 <p>Count: {state.count}</p>
 <button onClick={() => dispatch({ type: "increment" })}>+</
button>
 <button onClick={() => dispatch({ type: "decrement" })}>-</
button>
 <button onClick={() => dispatch({ type: "reset", payload: 0 })}
>Reset</button>
 </div>
 )
}
```

TYPING USEREF

DOM reference:

```
function Input() {
 const inputRef = useRef<HTMLInputElement>(null)

 function focus() {
 inputRef.current?.focus()
 }

 return (
 <>
 <input ref={inputRef} />
 <button onClick={focus}>Focus</button>
 </>
 )
}
```

Mutable value:

```
const countRef = useRef<number>(0)

countRef.current = 5
```

TYPING USECONTEXT

Create typed context:

```
interface ThemeContextType {
 theme: "light" | "dark";
 toggleTheme: () => void;
}

const ThemeContext = createContext<ThemeContextType |
null>(null)

function useTheme() {
 const context = useContext(ThemeContext)
 if (!context) {
 throw new Error("useTheme must be used within ThemeProvider")
 }
 return context
}

function ThemeProvider({ children }: { children: React.ReactNode
}) {
 const [theme, setTheme] = useState<"light" | "dark">("light")

 const toggleTheme = () => {
 setTheme(prev => prev === "light" ? "dark" : "light")
 }

 return (
 <ThemeContext.Provider value={{ theme, toggleTheme }}>
 {children}
 </ThemeContext.Provider>
```

```
 )
}
```

TYPING CUSTOM HOOKS

Custom hook with types:

```
function useLocalStorage<T>(key: string, initialValue: T) {
 const [value, setValue] = useState<T>(() => {
 const stored = localStorage.getItem(key)
 return stored ? JSON.parse(stored) : initialValue
 })

 const setStoredValue = (newValue: T | ((prev: T) => T)) => {
 setValue(prev => {
 const valueToStore = newValue instanceof Function
 ? newValue(prev)
 : newValue
 localStorage.setItem(key, JSON.stringify(valueToStore))
 return valueToStore
 })
 }

 return [value, setStoredValue] as const
}
```

Usage:

```
const [name, setName] = useLocalStorage<string>("name", "")
```

TYPING FORWARDREF

Forward ref to child:

```
interface InputProps {
 label: string;
}

const Input = forwardRef<HTMLInputElement, InputProps>(
 function Input({ label }, ref) {
 return (
 <label>
 {label}
 <input ref={ref} />
 </label>
 )
 }
)

function Form() {
 const inputRef = useRef<HTMLInputElement>(null)

 return <Input ref={inputRef} label="Name" />
}
```

GENERIC COMPONENTS

Component with type parameter:

```
interface ListProps<T> {
 items: T[];
 renderItem: (item: T) => React.ReactNode;
}

function List<T>({ items, renderItem }: ListProps<T>) {
 return (
 <ul>
 {items.map((item, index) => (
 <li key={index}>{renderItem(item)}</li>
 ))}
```

```
 </ul>
 )
}
```

Usage:

```
interface User {
 name: string;
 email: string;
}

function UserList({ users }: { users: User[] }) {
 return (
 <List
 items={users}
 renderItem={user => <span>{user.name}</span>}
 />
 )
}
```

DISCRIMINATED UNION PROPS

Component with variant props:

```
type ButtonProps =
 | { variant: "primary"; onClick: () => void }
 | { variant: "link"; href: string }

function Button(props: ButtonProps) {
 if (props.variant === "primary") {
 return <button onClick={props.onClick}>Click me</button>
 }
 return <a href={props.href}>Link</a>
}
```

Usage:

```
<Button variant="primary" onClick={() =>
console.log("clicked")} />
<Button variant="link" href="/about" />
```

RENDER PROPS PATTERN

Typed render prop:

```
interface RenderProps<T> {
 data: T;
 loading: boolean;
}

interface FetcherProps<T> {
 url: string;
 children: (props: RenderProps<T>) => React.ReactNode;
}

function Fetcher<T>({ url, children }: FetcherProps<T>) {
 const [data, setData] = useState<T | null>(null)
 const [loading, setLoading] = useState(true)

 useEffect(() => {
 fetch(url)
 .then(res => res.json())
 .then(data => {
 setData(data)
 setLoading(false)
 })
 }, [url])

 return <>{children({ data: data as T, loading })}</>
}
```

HIGHER ORDER COMPONENTS

Typed HOC:

```
interface WithLoadingProps {
 loading: boolean;
}

function withLoading<P extends object>(
 Component: React.ComponentType<P>
) {
 return function WithLoading(props: P & WithLoadingProps) {
 const { loading, ...rest } = props

 if (loading) {
 return <div>Loading...</div>
 }

 return <Component {...(rest as P)} />
 }
}
```

POLYMORPHIC COMPONENTS

Component that renders different elements:

```
type PolymorphicProps<E extends React.ElementType> = {
 as?: E;
 children: React.ReactNode;
} & React.ComponentPropsWithoutRef<E>

function Box<E extends React.ElementType = "div">({
 as,
 children,
 ...props
```

```
}: PolymorphicProps<E>) {
 const Component = as || "div"
 return <Component {...props}>{children}</Component>
}
```

Usage:

```
<Box>Default div</Box>
<Box as="span">Span element</Box>
<Box as="a" href="/about">Link element</Box>
```

PRACTICE

Create typed components.

Use typed hooks.

Create generic components.

Type event handlers.

Build a typed context.

Practice discriminated unions.

WHAT YOU LEARNED

You learned to type React components.

You learned to type hooks.

You learned event typing.

You learned generic components.

You learned advanced component patterns.

The next chapter teaches TypeScript programs.

BOOK 20: TYPESCRIPT ADVANCED

Chapter 8: TypeScript Programs

This chapter contains complete advanced TypeScript programs.

Each program demonstrates advanced concepts.

Study these to understand real-world patterns.

PROGRAM 1: TYPE-SAFE API CLIENT

Fully typed HTTP client with generics.

api-client.ts:

```
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE"

interface RequestConfig {
 headers?: Record<string, string>;
 timeout?: number;
}

interface ApiResponse<T> {
 data: T;
 status: number;
 headers: Headers;
}
```

```
interface ApiError {
 status: number;
 message: string;
 details?: unknown;
}

type Result<T, E = ApiError> =
 | { success: true; data: T }
 | { success: false; error: E }

interface ApiEndpoints {
 "/users": {
 GET: { response: User[]; params: { limit?: number } };
 POST: { response: User; body: CreateUserInput };
 };
 "/users/:id": {
 GET: { response: User; params: { id: string } };
 PUT: { response: User; body: UpdateUserInput; params: { id: string
} };
 DELETE: { response: void; params: { id: string } };
 };
 "/posts": {
 GET: { response: Post[]; params: { userId?: string } };
 POST: { response: Post; body: CreatePostInput };
 };
}

interface User {
 id: string;
 name: string;
 email: string;
 createdAt: string;
}

interface Post {
 id: string;
 userId: string;
 title: string;
 content: string;
```

```
}

type CreateUserInput = Omit<User, "id" | "createdAt">
type UpdateUserInput = Partial<CreateUserInput>
type CreatePostInput = Omit<Post, "id">

type ExtractParams<T extends string> =
 T extends `${infer _Start}:${infer Param}/${infer Rest}`
 ? Param | ExtractParams<`/${Rest}`>
 : T extends `${infer _Start}:${infer Param}`
 ? Param
 : never

class TypedApiClient {
 private baseUrl: string
 private defaultConfig: RequestConfig

 constructor(baseUrl: string, config: RequestConfig = {}) {
 this.baseUrl = baseUrl
 this.defaultConfig = config
 }

 private buildUrl(path: string, params?: Record<string,
unknown>): string {
 let url = this.baseUrl + path

 if (params) {
 Object.entries(params).forEach(([key, value]) => {
 url = url.replace(`:${key}`, String(value))
 })

 let queryParams = Object.entries(params)
 .filter(([key]) => !path.includes(`:${key}`))
 .map(([key, value]) => `${key}=
${encodeURIComponent(String(value))}`)
 .join("&")

 if (queryParams) {
 url += "?" + queryParams
 }
```

```
}

return url
}

async request<T>(
method: HttpMethod,
path: string,
options: {
params?: Record<string, unknown>;
body?: unknown;
config?: RequestConfig;
} = {}
): Promise<Result<T>> {
try {
let url = this.buildUrl(path, options.params)

let response = await fetch(url, {
method,
headers: {
"Content-Type": "application/json",
...this.defaultConfig.headers,
...options.config?.headers
},
body: options.body ? JSON.stringify(options.body) : undefined
})

if (!response.ok) {
return {
success: false,
error: {
status: response.status,
message: response.statusText
}
}
}

let data = await response.json() as T
return { success: true, data }
} catch (err) {
```

```
 return {
 success: false,
 error: {
 status: 0,
 message: err instanceof Error ? err.message : "Unknown error"
 }
 }
 }
 }

 get<P extends keyof ApiEndpoints>(
 path: P,
 params?: ApiEndpoints[P]["GET"] extends { params: infer Params }
 ? Params
 : never
 ): Promise<Result<ApiEndpoints[P]["GET"]["response"]>> {
 return this.request("GET", path, { params: params as
Record<string, unknown> })
 }

 post<P extends keyof ApiEndpoints>(
 path: P,
 body: ApiEndpoints[P]["POST"] extends { body: infer Body } ?
Body : never
 ): Promise<Result<ApiEndpoints[P]["POST"]["response"]>> {
 return this.request("POST", path, { body })
 }
}

async function main() {
 let api = new TypedApiClient("https://api.example.com")

 let usersResult = await api.get("/users", { limit: 10 })

 if (usersResult.success) {
 usersResult.data.forEach(user => {
 console.log(user.name)
 })
 }
```

```
 let createResult = await api.post("/users", {
 name: "Ali",
 email: "ali@example.com"
 })

 if (createResult.success) {
 console.log("Created user:", createResult.data.id)
 }
}
```

PROGRAM 2: STATE MACHINE

Type-safe finite state machine.

state-machine.ts:

```
type StateConfig<
 States extends string,
 Events extends string
> = {
 [S in States]: {
 on?: {
 [E in Events]?: States;
 };
 entry?: () => void;
 exit?: () => void;
 };
}

type ExtractStates<T> = T extends StateConfig<infer S, any> ?
S : never
type ExtractEvents<T> = T extends StateConfig<any, infer E> ?
E : never

class StateMachine<
 Config extends StateConfig<string, string>,
```

```
State extends ExtractStates<Config> = ExtractStates<Config>,
Event extends ExtractEvents<Config> = ExtractEvents<Config>
> {
private currentState: State
private config: Config
private listeners: Set<(state: State) => void> = new Set()

constructor(config: Config, initialState: State) {
this.config = config
this.currentState = initialState
this.executeEntry(initialState)
}

private executeEntry(state: State): void {
let stateConfig = this.config[state]
if (stateConfig?.entry) {
stateConfig.entry()
}
}

private executeExit(state: State): void {
let stateConfig = this.config[state]
if (stateConfig?.exit) {
stateConfig.exit()
}
}

getState(): State {
return this.currentState
}

can(event: Event): boolean {
let stateConfig = this.config[this.currentState]
return stateConfig?.on?.[event] !== undefined
}

send(event: Event): State {
let stateConfig = this.config[this.currentState]
let nextState = stateConfig?.on?.[event] as State | undefined
```

```
 if (nextState && nextState !== this.currentState) {
 this.executeExit(this.currentState)
 this.currentState = nextState
 this.executeEntry(nextState)
 this.notifyListeners()
 }

 return this.currentState
 }

 subscribe(listener: (state: State) => void): () => void {
 this.listeners.add(listener)
 return () => this.listeners.delete(listener)
 }

 private notifyListeners(): void {
 this.listeners.forEach(listener => listener(this.currentState))
 }
}

type TrafficLightStates = "green" | "yellow" | "red"
type TrafficLightEvents = "timer" | "emergency"

let trafficLightConfig: StateConfig<TrafficLightStates,
TrafficLightEvents> = {
 green: {
 on: {
 timer: "yellow",
 emergency: "red"
 },
 entry: () => console.log("Light is GREEN - Go!"),
 exit: () => console.log("Leaving green state")
 },
 yellow: {
 on: {
 timer: "red",
 emergency: "red"
 },
 entry: () => console.log("Light is YELLOW - Slow down!"),
 exit: () => console.log("Leaving yellow state")
```

```
 },
 red: {
 on: {
 timer: "green"
 },
 entry: () => console.log("Light is RED - Stop!"),
 exit: () => console.log("Leaving red state")
 }
}

let trafficLight = new StateMachine(trafficLightConfig, "red")

trafficLight.subscribe(state => {
 console.log("State changed to:", state)
})

trafficLight.send("timer")
trafficLight.send("timer")
trafficLight.send("timer")
trafficLight.send("emergency")
```

PROGRAM 3: DEPENDENCY INJECTION CONTAINER

Type-safe IoC container.

di-container.ts:

```
type Constructor<T = any> = new (...args: any[]) => T

interface ServiceDefinition<T> {
 factory: () => T;
 singleton: boolean;
}

type ServiceKey = string | symbol
```

```
class Container {
 private services = new Map<ServiceKey,
ServiceDefinition<unknown>>()
 private instances = new Map<ServiceKey, unknown>()

 register<T>(
 key: ServiceKey,
 factory: () => T,
 options: { singleton?: boolean } = {}
 ): this {
 this.services.set(key, {
 factory,
 singleton: options.singleton ?? true
 })
 return this
 }

 registerClass<T>(
 key: ServiceKey,
 ctor: Constructor<T>,
 options: { singleton?: boolean } = {}
 ): this {
 return this.register(key, () => new ctor(), options)
 }

 get<T>(key: ServiceKey): T {
 let definition = this.services.get(key)

 if (!definition) {
 throw new Error(`Service not found: ${String(key)}`)
 }

 if (definition.singleton) {
 if (!this.instances.has(key)) {
 this.instances.set(key, definition.factory())
 }
 return this.instances.get(key) as T
 }

 return definition.factory() as T
```

```
 }

 has(key: ServiceKey): boolean {
 return this.services.has(key)
 }
}

type TypedContainer<Services extends Record<ServiceKey,
unknown>> = {
 register<K extends keyof Services>(
 key: K,
 factory: () => Services[K],
 options?: { singleton?: boolean }
 ): TypedContainer<Services>;

 get<K extends keyof Services>(key: K): Services[K];
}

function createTypedContainer<
 Services extends Record<ServiceKey, unknown>
>(): TypedContainer<Services> {
 return new Container() as unknown as TypedContainer<Services>
}

interface Logger {
 log(message: string): void;
}

interface Database {
 query(sql: string): Promise<unknown[]>;
}

interface UserService {
 getUser(id: string): Promise<{ id: string; name: string }>;
}

interface AppServices {
 logger: Logger;
 database: Database;
 userService: UserService;
```

```
}

let container = createTypedContainer<AppServices>()

container.register("logger", () => ({
 log(message: string) {
 console.log("[LOG]", message)
 }
}))

container.register("database", () => ({
 async query(sql: string) {
 console.log("Executing:", sql)
 return []
 }
}))

container.register("userService", () => {
 let logger = container.get("logger")
 let db = container.get("database")

 return {
 async getUser(id: string) {
 logger.log("Getting user: " + id)
 await db.query("SELECT * FROM users WHERE id = " + id)
 return { id, name: "Ali" }
 }
 }
})

async function main() {
 let userService = container.get("userService")
 let user = await userService.getUser("123")
 console.log("User:", user)
}

main()
```

PROGRAM 4: EVENT BUS

Typed pub/sub system.

event-bus.ts:

```
type EventMap = Record<string, unknown>

type EventCallback<T> = (data: T) => void | Promise<void>

interface Subscription {
 unsubscribe(): void;
}

class TypedEventBus<Events extends EventMap> {
 private handlers = new Map<keyof Events,
Set<EventCallback<unknown>>>()

 on<K extends keyof Events>(
 event: K,
 callback: EventCallback<Events[K]>
 ): Subscription {
 if (!this.handlers.has(event)) {
 this.handlers.set(event, new Set())
 }

 let handlers = this.handlers.get(event)!
 handlers.add(callback as EventCallback<unknown>)

 return {
 unsubscribe: () => {
 handlers.delete(callback as EventCallback<unknown>)
 }
 }
 }

 once<K extends keyof Events>(
 event: K,
```

```
 callback: EventCallback<Events[K]>
 ): Subscription {
 let subscription: Subscription

 let wrapper: EventCallback<Events[K]> = async (data) => {
 subscription.unsubscribe()
 await callback(data)
 }

 subscription = this.on(event, wrapper)
 return subscription
 }

 async emit<K extends keyof Events>(
 event: K,
 data: Events[K]
 ): Promise<void> {
 let handlers = this.handlers.get(event)

 if (handlers) {
 let promises = Array.from(handlers).map(handler =>
 Promise.resolve(handler(data))
 )
 await Promise.all(promises)
 }
 }

 off<K extends keyof Events>(event: K): void {
 this.handlers.delete(event)
 }

 listenerCount<K extends keyof Events>(event: K): number {
 return this.handlers.get(event)?.size ?? 0
 }
}

interface AppEvents {
 "user:login": { userId: string; timestamp: Date };
 "user:logout": { userId: string };
 "order:created": { orderId: string; total: number };
```

```
 "order:completed": { orderId: string };
 "notification:send": { userId: string; message: string };
}

let eventBus = new TypedEventBus<AppEvents>()

eventBus.on("user:login", async (data) => {
 console.log("User logged in:", data.userId, "at", data.timestamp)
})

eventBus.on("order:created", async (data) => {
 console.log("New order:", data.orderId, "Total:", data.total)

 await eventBus.emit("notification:send", {
 userId: "admin",
 message: "New order received: " + data.orderId
 })
})

eventBus.on("notification:send", async (data) => {
 console.log("Sending notification to", data.userId + ":",
data.message)
})

async function main() {
 await eventBus.emit("user:login", {
 userId: "user123",
 timestamp: new Date()
 })

 await eventBus.emit("order:created", {
 orderId: "order456",
 total: 99.99
 })
}

main()
```

PROGRAM 5: VALIDATION LIBRARY

Schema-based type-safe validation.

validator.ts:

```
type ValidationResult<T> =
 | { success: true; data: T }
 | { success: false; errors: string[] }

interface Schema<T> {
 parse(value: unknown): ValidationResult<T>;
 optional(): Schema<T | undefined>;
}

function string(): Schema<string> {
 return {
 parse(value: unknown): ValidationResult<string> {
 if (typeof value === "string") {
 return { success: true, data: value }
 }
 return { success: false, errors: ["Expected string"] }
 },
 optional() {
 return optional(this)
 }
 }
}

function number(): Schema<number> {
 return {
 parse(value: unknown): ValidationResult<number> {
 if (typeof value === "number" && !isNaN(value)) {
 return { success: true, data: value }
 }
 return { success: false, errors: ["Expected number"] }
 },
 optional() {
```

```
 return optional(this)
 }
 }
}

function boolean(): Schema<boolean> {
 return {
 parse(value: unknown): ValidationResult<boolean> {
 if (typeof value === "boolean") {
 return { success: true, data: value }
 }
 return { success: false, errors: ["Expected boolean"] }
 },
 optional() {
 return optional(this)
 }
 }
}

function optional<T>(schema: Schema<T>): Schema<T |
undefined> {
 return {
 parse(value: unknown): ValidationResult<T | undefined> {
 if (value === undefined) {
 return { success: true, data: undefined }
 }
 return schema.parse(value)
 },
 optional() {
 return this
 }
 }
}

function array<T>(itemSchema: Schema<T>): Schema<T[]> {
 return {
 parse(value: unknown): ValidationResult<T[]> {
 if (!Array.isArray(value)) {
 return { success: false, errors: ["Expected array"] }
 }
```

```
 let results: T[] = []
 let errors: string[] = []

 value.forEach((item, index) => {
 let result = itemSchema.parse(item)
 if (result.success) {
 results.push(result.data)
 } else {
 errors.push(...result.errors.map(e => `[${index}]: ${e}`))
 }
 })

 if (errors.length > 0) {
 return { success: false, errors }
 }

 return { success: true, data: results }
 },
 optional() {
 return optional(this)
 }
 }
}

type InferSchema<T> = T extends Schema<infer U> ? U : never

type ObjectShape = Record<string, Schema<unknown>>

type InferObject<T extends ObjectShape> = {
 [K in keyof T]: InferSchema<T[K]>
}

function object<T extends ObjectShape>(shape: T):
Schema<InferObject<T>> {
 return {
 parse(value: unknown): ValidationResult<InferObject<T>> {
 if (typeof value !== "object" || value === null) {
 return { success: false, errors: ["Expected object"] }
 }
```

```
 let obj = value as Record<string, unknown>
 let result: Record<string, unknown> = {}
 let errors: string[] = []

 for (let key in shape) {
 let fieldResult = shape[key].parse(obj[key])
 if (fieldResult.success) {
 result[key] = fieldResult.data
 } else {
 errors.push(...fieldResult.errors.map(e => `${key}: ${e}`))
 }
 }

 if (errors.length > 0) {
 return { success: false, errors }
 }

 return { success: true, data: result as InferObject<T> }
 },
 optional() {
 return optional(this)
 }
 }
}

let userSchema = object({
 name: string(),
 email: string(),
 age: number().optional(),
 tags: array(string())
})

type User = InferSchema<typeof userSchema>

let result1 = userSchema.parse({
 name: "Ali",
 email: "ali@example.com",
 age: 25,
 tags: ["developer", "typescript"]
```

```
})

if (result1.success) {
 console.log("Valid user:", result1.data)
 console.log("Name:", result1.data.name)
}

let result2 = userSchema.parse({
 name: 123,
 email: "test@example.com",
 tags: "not-an-array"
})

if (!result2.success) {
 console.log("Validation errors:", result2.errors)
}
```

WHAT YOU LEARNED

You learned advanced TypeScript patterns.

You learned type-safe API design.

You learned state machines with types.

You learned dependency injection.

You learned validation with type inference.

BOOK SERIES COMPLETE

Congratulations!

You have completed the JavaScript For Noobs book series.

You started with zero programming knowledge.

You learned JavaScript fundamentals.

You learned DOM manipulation and events.

You learned async programming and APIs.

You learned Node.js and Express.

You learned React for building interfaces.

You learned Three.js for 3D graphics.

You learned TypeScript for type safety.

You are now ready to build real applications.

Keep learning, keep building, keep growing.